LLNL-CONF-493773

A Topology-Aware Application-Independent Load Model for Parallel Codes

O. T. Pearce, B. R. de Supinski, G. T. Gamblin, M. W. Schulz, N. M. Amato

August 17, 2011

17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming
New Orleans, LA, United States
February 25, 2012 through February 29, 2012

A Topology-Aware Application-Independent Load Model for Parallel Codes

Olga Pearce
Texas A&M University, Lawrence Livermore National Laboratory
olga@cse.tamu.edu, olga@llnl.gov

Todd Gamblin
Bronis R. de Supinski
Martin Schulz
Lawrence Livermore National Laboratory
tgamblin,bronis,schulzm@llnl.gov

Nancy M. Amato
Texas A&M University
amato@cse.tamu.edu

Abstract

Load balance is critical for performance in large parallel applications. An imbalance on today's fastest supercomputers can force hundreds of thousands of cores to idle, and on future exascale machines this cost will increase over a thousand fold. Rectifying imbalance requires detailed understanding of the amount of computational load per process *and* an application's simulated domain, but no existing algorithms sufficiently account for both factors. Load balancers are either a) integrated into applications *ad hoc*, making implicit assumptions about the load, or b) oblivious to application semantics and unable to make informed rebalancing decisions.

We present a novel application-independent load model that also captures application topology, load, and execution time. Developers use abstract work units to inform our model of their load metrics. Using this abstraction, we develop a cost model for load imbalance. We show that our model can be used both to select the most effective load metrics for particular applications and to compare the effectiveness of load balancing algorithms in particular imbalance scenarios. Our results show that failing to rebalance can slow execution 2.15 times, and that choosing the wrong load balancing algorithm can make execution 2.17 times slower.

1. Introduction

Optimizing high-performance physical simulations to run on ever-growing supercomputing hardware is challenging. The largest modern parallel simulation codes are written using synchronous message passing frameworks such as MPI, and dynamic behavior in such applications may lead to imbalances in computational load among processors. Even a single slow task can force all others to wait. On modern machines with hundreds of thousands or more processors, this cost can be enormous. Future machines will support even more parallel tasks, and efficiently redistributing and balancing load among them will be critical for good performance.

Prior work has explored tools to measure large-scale computational load [16, 32] efficiently. These tools can provide insight into the source location that caused an imbalance [31] and into the distribution of the load, but this knowledge alone is not sufficient to correct the load. Existing load measurements do not account for constraints on rebalancing imposed by the topology of the simulated physical domain. Real physical applications may only be able to rebalance load by moving simulated entities between nearby processors. Because this requires knowledge of application topology, load metrics that consider only MPI ranks or statistical metrics are unable to guide the correction of the imbalance. As a result, many applications resort to custom load balancing schemes and build their own model of the application workload to guide the assignment of work to processes. In these custom schemes, application developers estimate the costs of the computation, but their estimates typically capture only the developer's best guess and often do not correspond to the actual computational costs.

To address this challenge, we develop an abstract computational load model and novel load metrics that account for application topology and connectivity between processes. We identify a set of metrics to evaluate the accuracy of our model, and we use our load model to evaluate the cost of correcting a load imbalance with particular load balancing algorithms. We demonstrate this methodology on two large-scale production applications: a molecular dynamics code and a dislocation dynamics simulation. We use this model to compare load balancing methods and to select the method that most efficiently balances a particular application state.

Specifically, we make the following contributions:

- A novel application-independent load model that captures an application's topology, load, and execution time;
- A methodology to evaluate load imbalance and the efficiency with which particular load balance schemes correct it;
- Techniques to evaluate application-provided models and to compare candidate application models;
- An extensive evaluation of load balance characteristics and models for two large-scale production simulations;
- A cost model to evaluate available balancing mechanisms and to select the one most efficient for a particular imbalance scenario.

Our experiments show that *ad hoc* application models can mispredict load imbalance by as much as 70%, misrepresenting load distribution and the cost of rebalancing. We also show that a widely used measure of load imbalance, the ratio of maximum load to average load, along with other statistical metrics, inaccurately represent load distributions in the context of balancing algorithms, because they do not consider the application topology. Alternatively, our models provide insight into the cost of different algorithms such as diffusion [10] and repartitioning [26]. We validate our cost models

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 (LLNL-CONF-493773).

and demonstrate that failing to rebalance can lengthen execution by a factor of 2.15, while choosing the wrong load balancing algorithm can make runtime 2.17 times longer.

The remainder of this paper is organized as follows. We give an overview of our method in Section 2 and demonstrate the shortfalls of current load metrics in Section 3. We define our topology-aware application-independent load model in Section 4 and our cost model for load balancing algorithms in Section 5. We describe our target applications and their balancing algorithms in Section 6. We evaluate application models and demonstrate how to use our load model to select the appropriate load balancing algorithm in Section 7.

2. Overview of Approach

High-performance physical simulations need guidance on *when* and *how* to balance their computational load. Existing load metrics have focused on measuring load on processes over time and on tracing load imbalances to particular regions of application source code. While these tools are useful for locating imbalances, they cannot be used to estimate the cost of load balancing because the speed with which work can be redistributed depends on its layout in the simulated domain.

This paper presents a novel load model that can be used to evaluate the cost of correcting an imbalance and to guide correction of the imbalance at runtime. To guide the application on *how* to correct the imbalance, our load model uses a graph to represent the application's topology, where application elements correspond to vertices and edges are used to indicate dependencies or communication between elements. Our model uses existing tools' measurements of the *degree* of imbalance and adds the ability to evaluate the load in terms of application topology. An application developer needs only to provide an abstract graph of work units and their communications as input, and our framework will build a graph to represent this topology in an application-independent manner. We show that topology-awareness is essential in appropriately evaluating the imbalance and mechanisms to correct it.

These techniques comprise a general framework for characterizing load imbalance in large-scale applications, and they augment existing load metrics by facilitating the evaluation of developer-provided load estimation schemes. Thus, an application developer can use them to refine *ad hoc* load models and to understand their limitations. We demonstrate this process for two large-scale production applications in Section 7.1. The developer can then use our cost model approach to select from available load balancing algorithms, as we show in Section 7.2.

3. Deficiencies of Conventional Load Metrics

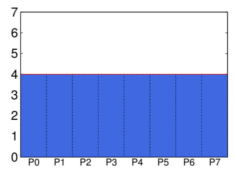
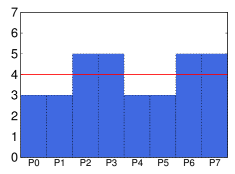
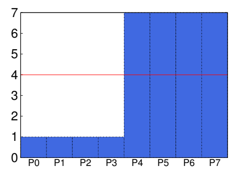
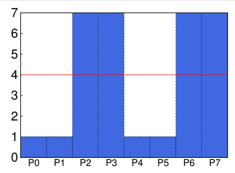
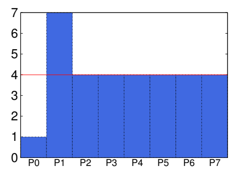
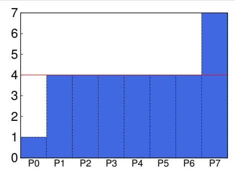
Load imbalance is formally defined as an uneven distribution of work, or computational *load*, among tasks in a parallel system. In large-scale SPMD applications with synchronous time steps, load balance can be costly because all processes are forced to wait at synchronization points for the most overloaded process. The performance penalty for such a load imbalance grows linearly as the number of processors increases, so it is particularly important to continuously balance large-scale synchronous simulations as their load distribution evolves over the course of a run.

Load balance metrics are used to characterize how unevenly work is distributed among proceses. The most commonly used load balance metric is *percent imbalance*, λ:

$$\lambda = \left(\frac{L_{max}}{\overline{L}} - 1 \right) \times 100\% \tag{1}$$

where L_{max} is the maximum load on any process and $\overline{L}$ is the mean of the loads of all processes in the job. This metric measures the performance lost to imbalanced load or, conversely, the performance that could be reclaimed by balancing the load. Percent imbalance measures the *severity* of load imbalance. However, it ignores the application's data model, its topology, and statistical properties of the load distribution. These properties can provide insight into how quickly a particular algorithm can correct an imbalance.

Table 1. Examples of load distributions and their moments

	Load on each Process	$\overline{L}$	λ	σ	g_1	g_2
(a)		4	0%	0	0	0
(b)		4	25%	1	1	-2
(c)		4	75%	3	1	-2
(d)		4	75%	3	1	-2
(e)		4	75%	1.5	2	1
(f)		4	75%	1.5	2	1

Statistical moments can provide a more detailed picture of load distribution. These are aggregate metrics, and they can indicate whether a distribution features a few highly loaded outliers or many slightly imbalanced processes. These properties directly affect the type of balancing algorithm that will most efficiently correct the imbalance. Small imbalances can be faster to correct with diffusive algorithms [10] while the presence of an outlier in the load distribution may require more drastic, global corrections.

We focus on the three most common statistical moments, standard deviation σ, skewness g_1 and kurtosis g_2:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=0}^{n} (L_i - \overline{L})^2} \tag{2}$$

$$g_1 = \frac{\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^3}{\left(\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^2\right)^{3/2}} \tag{3}$$

$$g_2 = \frac{\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^4}{\left(\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^2\right)^{2}} - 3 \tag{4}$$

where n is the number of processes and L_i is the load on the i^{th} process. Positive skewness means that relatively few processes have higher than average load, while negative skewness means that relatively few processes have lower than average load. A normal distribution of load implies skewness of 0. Higher kurtosis means that more of the variance arises from infrequent extreme deviations, while lower kurtosis corresponds to frequent modestly sized deviations. A normal distribution has kurtosis of 0.

Statistical moments provide interesting information about load distribution, but they are still insufficient to evaluate the speed with which a load balance can be corrected because they do not include information about the proximity of physical elements in the simulation space.

Table 1 illustrates this using several load distributions for which the statistical moments fail to distinguish key properties. For simplicity, the examples show a one-dimensional topology of eight processes $P_0 \ldots P_7$ where P_i and P_{i+1} perform computation on neighboring domains. The figure shows that load metrics cannot distinguish cases (e) and (f), while the difficulty of correcting these load scenarios would vary greatly in a physical simulation in which the computation is optimal when neighboring portions of the simulated space are assigned to the neighboring processes. In case (e), we could simply move the extra load on P_1 to P_0, while in (f) the extra load from P_7 would first need to displace work to P_6, P_5, and so on through P_1 until the underloaded P_0 receives enough work.

4. Topology-Aware Load Model

The observations in Section 3 show that we require a load model that is aware of the application's topology in order to understand the load imbalance present in the application and, more importantly, how to correct it. A model that does not include the application topology will fail to capture the critical impact of the proximity of elements in the simulation space and the mapping of the simulation space onto the process space. Our investigation of large-scale scientific applications has driven the development of a novel topology-aware, application-independent load model that represents the units of work in the application and the communication and dependencies between them. Our work provides a general methodology to represent application topology and to map observed application performance accurately to the model elements that constitute the application units of work.

Parallel scientific applications must decompose their physical domain into work units, which, in different applications, can be units of the simulated physical space, particles modeled, or random samples performed on the domain. Some work units may involve more or less computation than others due to, e.g., their spatial proximity to other work units or their location within the simulated physical space (e.g., at its boundaries). Most load balancing algorithms analyze and redistribute work using the same granularity as the application's domain decomposition. We use this granularity for our load model so that it reflects load-balanced work units, their communication and dependencies, and their mapping to processes.

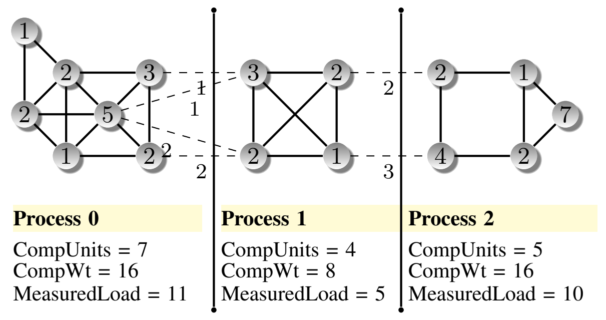


Figure 1. Topology Aware Load Model.

Our load model expresses the application's topology as a graph in which the nodes are migratable units of computation within the application with the appropriate granularity, and the edges represent communication and/or dependencies between them.

Figure 1 illustrates our load model: the edges represent bidirectional interactions between work units. Solid edges represent interactions within a process, while dashed edges represent interprocess communication. The relationships between application work units within the domain decomposition provide the communication structure and the relative weights of computation in the model. Node weights indicate the computation required for each unit of work as anticipated by the application (i.e., the application load model). Importantly, we can correlate this information to wall-clock measurements of the load on each process. The example in Figure 1 shows that Process 0 has 7 work units with an application anticipated load or relative computation weight of 16, and its work units have 4 channels of communication with work units on Process 1 with a total relative communication cost of 6. We measure the load on Process 0 to be 11.

The difference in modeled load and measured load should be carefully considered; if the model is accurate, there is a linear relationship between the two which can be easily corrected by scaling. If they are not directly proportional, the application model is incomplete and could be improved. We discuss our methodology on application model evaluation in Section 7.1.

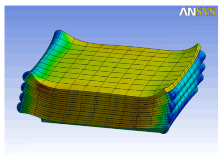
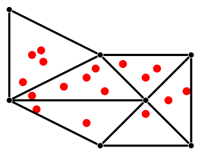
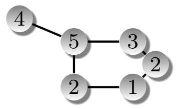
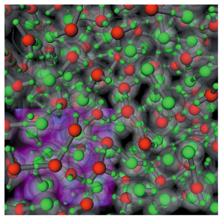
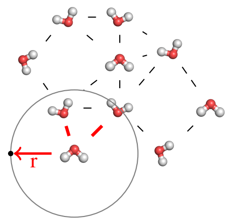
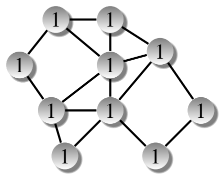
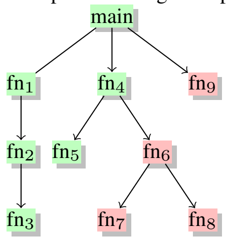
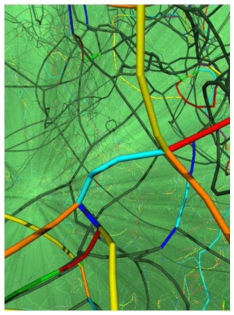
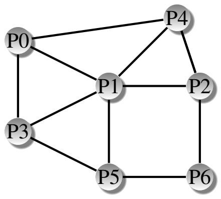
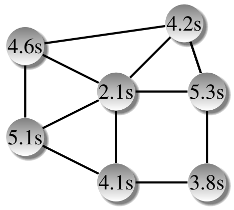
When we are satisfied with the model accuracy, we can use the model to compute the load distribution metrics and to observe how the load is distributed throughout the process space in a manner meaningful in terms of application topology.

To ensure that our model is application-independent, we use this graph abstraction to represent application load distributions. Thus, to use our framework, application developers must provide a mapping their own work units to our abstraction. To make this process easy for application developers, we developed an API that allows a developer to identify the application work units and the application model of the computation as well as the communication among work units. Application developers can easily locate the needed information because they are aware of the explicit and implicit work that the application performs. We require the application to provide the information in the distributed compressed storage format (CSR), which enables efficient storage of this typically very sparse information, as well as interoperability with partitioning tools such as ParMetis [25].

Table 2 illustrates the versatility of our model by showing work unit mappings for three major types of scientific applications.

Unstructured Mesh. In unstructured mesh applications, each cell in the mesh is a unit of work. We represent the mesh connectivity with edges. In some unstructured mesh applications, the cells may require similar computation and we would anticipate unit compu-

Table 2. Applications and their representation in our Load Model

Type of Application, work units and connectivity	Sample Application Image	Representation in App.	Our Representation
(a) Unstructured Mesh • e.g., particle transport or finite element applications • work units: cell volume or number of samples in each cell (Monte Carlo algorithms) • connectivity: mesh connectivity			
(b) N-body • e.g., Molecular Dynamics Applications • work units: (sampled) molecules • connectivity: molecules within range of interaction r (as defined by the application)			
(c) Other - Empirical Model • e.g., ParaDiS (Section 6.2) • connectivity: graph of process communication • work units: time in developer-defined 'key' routines (green); incomplete coverage of application behavior (red)  main fn$_1$ fn$_4$ fn$_9$ fn$_2$ fn$_5$ fn$_6$ fn$_3$ fn$_7$ fn$_8$			

tation per mesh cell. In others, the computation per cell may be proportional to the cell's volume, and we reflect this in the weight of each node in our model. Table 2(a) shows an unstructured mesh application that performs a Monte Carlo algorithm on its mesh. In this case, the work performed is proportional to the number of samples in each mesh cell, so we use the sample count as the node weight. Per our model, communication operations between neighboring grid cells are shown as edges.

Molecular Dynamics. In classical molecular dynamics applications and other N-body simulations, each individual body is a unit of work. Edges reflect the simulated neighborhood of the bodies: each body is connected to others within a cutoff radius (i.e., those with which it interacts), as illustrated in Table 2(b). As we discuss in Section 7.1, we can select from several models for computation per work unit. Simple models assume that the work per body is constant, while others reflect the density of the body's simulated neighborhood.

Empirical Model. Other applications, such as the dislocation dynamics code ParaDiS [8], use empirical models to anticipate computation per unit of work. An application developer can construct this type of model by placing timers around important regions of computation regions. Table 2(c) shows how *ad hoc* placement of timers may omit important load constituents.

5. Modeling the Cost of Load Balancing

In this section, we use our abstract graph-based load model to evaluate the cost of balancing for two common load balancing algorithms for physical simulations. Our cost model can be used to select the most efficient algorithm for particular imbalance scenarios.

5.1 Types of Load Balancing Algorithms

Global Algorithms. A global balancing algorithm [12, 26, 34] takes information about the load of all tasks in the parallel application and decides how to redistribute load evenly in a single step. Global decisions can be costly, as sequential implementations must process data for an entire parallel system, and parallel implementations may require excessive data movement. However, if the cost of balancing is not high, global algorithms may balance load in a single step, and local minimum and maximum loads are handled correctly.

Diffusive Algorithms. A diffusive balancing algorithm [9, 10] performs local corrections at each step, and only moves work units within a local neighborhood in the logical simulation topology. Diffusive algorithms may take many steps to rectify a large imbalance, because load can only move a limited distance in the simulation space. However, diffusive algorithms are scalable because they only require local information, and data movements can be mapped to perform well on high diameter mesh and torus networks used in the largest machines, because they reward communication locality.

Although we do not investigate them in this paper, our methodology can apply to other types of balancing algorithms. *Hybrid balancing algorithms* use a global scheme to correct drastic imbalances and a diffusion scheme to avoid subsequent local imbalances. *Hierarchical balancing algorithms* apply some balancing

Algorithm 5.1 Diffusion algorithm [10]

```
Input. $L_i \leftarrow$ load of process $i$
    $D_i \leftarrow$ neighborhood of process $i$, defined in Load Model graph
    $L_{ij} \leftarrow$ load of process $j \in D_i$
    $\gamma \leftarrow$ coefficient for how much load can be moved in one timestep
    $\Delta L_i \leftarrow$ change in load of process $i$ from prev. to curr. iteration
    steps $\leftarrow 0$
1: All processes in parallel do
2: while imbalance > threshold do
3:     $L_i = L_i + \sum_{j \in D_i} \gamma(L_i - L_{ij})$
4:     steps++
5: end while
```

algorithm to a hierarchy of process partitions, using divide-and-conquer to reduce the complexity of the algorithm.

5.2 Cost Model for Load Balancing Algorithms

In this section we develop a cost model that accounts for the rebalancing characteristics of diffusive and global algorithms in the context of our abstract load graph. Application developers typically choose balancers based on their intuitions about the scalability of particular algorithms. For example, one might expect the cost of a global balancing scheme to be higher than that of a diffusive algorithm at scale because the time required for an immediate rebalance outweighs the amortized cost of local diffusive balancing. Our cost model allows us to validate these intuitions and to select the most efficient algorithm quantitatively.

The cost of a load balancing algorithm is the combined cost of making a load balancing decision and the time required for moving data to redistribute work load:

$$C_{BalAlgo} = C_{LbDecision} + C_{DataMvmt} \tag{5}$$

where $BalAlgo$ can be either $global$ or $diffusion$. Additionally, load balancers that take several steps to bring the application to a balanced state incur the cost of running the application in an imbalanced state during those steps. To provide an accurate comparison of the costs of global and diffusive load balancing methods, we define the total cost of a load balancer as the application time required to perform these steps while using it.

We define the *number of steps to converge (steps)* of a diffusion-like load balancing scheme as the number of iterations required to converge to a balanced load distribution. Given a load balance algorithm, we need a way to estimate how many steps it will take before converging. To do this, we apply Algorithm 5.1, using our topology-aware model of the initial load balance distribution as input. The algorithm simulates the movement of load through the topology of the application, using any constraints defined for particular balancers. Specifically, we define a coefficient γ to model the amount of load that can be moved in one time step. Our technique accounts for local minima and maxima because it moves the simulated load through the simulated network as the actual diffusive algorithm would.

We can estimate *steps* of a diffusion algorithm using this simulation much more quickly than we can perform the actual diffusion. Further, we can evaluate its cost without perturbing the application. If the simulation predicts that diffusion will take too long, we can use a different load balancing algorithm such as a global load balancing scheme.

We define the amount of data that a diffusion algorithm moves as the sum of the data moved in each of its steps:

$$DataMoved = \sum_{s=0}^{steps} \sum_{i=0}^{procs} \sum_{j \in D_i} \gamma(L_i - L_{ij}) \tag{6}$$

We calculate the data movement simultaneously with our convergence simulation. As discussed in Section 3, the data movement costs of a diffusion scheme depend on the topological relationship of the load, which we model through iterative simulation.

We compute the total time the application takes when load balancing via a diffusion algorithm as:

$$AppTime_{diffusion} = \sum_{i=0}^{steps} (C_{diffusion_i} + L_{max_i}) \tag{7}$$

where L_{max_i} is the maximum process load at step *i*, and $C_{diffusion_i}$ is the cost of the diffusion algorithm at step i from Equation 5. L_{max_i} to accounts for the time lost due to imbalance before the algorithm has completely converged.

Let α be a function of the step index i describing the amount by which a diffusion algorithm reduces L_{max_i}. This function can be defined to reflect the behavior of the specific application or deduced from previous runs, and could be constant or time-varying, e.g. a compound rate. We can approximate the total application time for a diffusion scheme as follows:

$$AppTime_{diffusion} \approx steps \times C_{diffusion} + \sum_{i=0}^{steps} \alpha(i) \times L_{max} \tag{8}$$

For a global scheme, we compute the total application time as:

$$AppTime_{global} = C_{global} + \sum_{i=0}^{steps} L_{max_i} \tag{9}$$

where C_{global} is the cost of the global balancing algorithm, from Equation 5. For comparison purposes, we assume that the global load balancing algorithm is invoked only once during the time that the diffusion algorithm steps to reach overall load balance.

The above equations assume that, once balanced, the application does not again become imbalanced. In reality, physics and other effects in the simulated domain may effect the load. For many applications, we can model this using a simple rate.

Let β be the dual of α: a function of i describing the amount the application becomes *imbalanced* on step i. Similarly to α, β can be specified for a particular application or problem set. We approximate the application time by:

$$AppTime_{global} \approx C_{global} + \sum_{i=0}^{steps} \beta(i) \times L_{max} \tag{10}$$

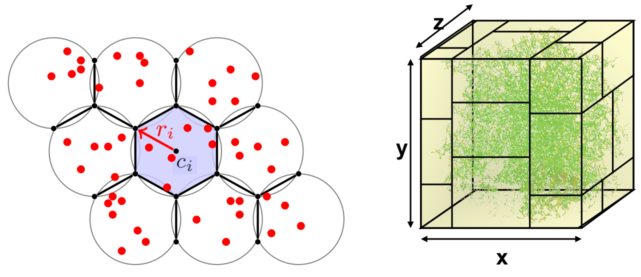
where L_{max} is the maximum process load immediately after the the global balancing algorithm completes. When $\beta(i)$ is large, load diverges quickly after global balancing steps, and the lowest cost option may be a hybrid algorithm. This will apply a global balancer when drastic changes are necessary and it will apply a diffusion scheme periodically to keep more modest imbalances in check. The cost of this hybrid algorithm can be assessed by combining the above equations. Our cost model would allow us to adjust the frequency with which each is applied.

6. Applications

To evaluate our load and cost models, we conducted experiments with two large-scale scientific applications, ddcMD and ParaDiS. Here, we describe the applications, their data models, and their associated load balancing algorithms.

6.1 ddcMD

ddcMD [11, 29] is a highly optimized molecular dynamics application that has twice won the Gordon Bell prize for high performance computing [15, 30]. It is written in C and uses the MPI library for communication between processors. In the ddcMD model, each process owns a subset of the simulated particles and maintains lists of other particles with which its particles interact.



(a) ddcMD. (b) ParaDiS.

Figure 2. Domain decomposition in ddcMD and ParaDiS.

To allocate particles to processes, ddcMD uses a *Voronoi* domain decomposition. That is, each process is assigned a point as its center, and it then "owns" the particles that are nearer to its center than any other. A *Voronoi cell* is the set of *all* points nearest to a particular center. Figure 2(a) shows a sample decomposition, with cells outlined in black and with particles shown in red. ddcMD also maintains a *bounding sphere* around its cell, where the sphere is defined by a radius calculated as the maximum distance of any atom in the domain to its center.

During execution, atoms owned by a process may move outside of their cell. When this happens, ddcMD uses a built-in diffusion load balancer that uses a load particle density gradient calculation to reassign load. The balancer does this by moving the Voronoi centers so that the walls of the Voronoi cells shift towards regions of greater density. Voronoi centers are limited in how much closer to a neighboring cell they can move, and the decompositions possible with this scheme are limited by Voronoi constraints on the shape of cells. To represent work units to the load balancer, ddcMD uses three application-specific models:

1. *Molecules*: number of particles (molecules) per process;
2. *Barriers*: time each process spends outside of barriers;
3. *Forces*: time spent calculating interactions on each process.

For our global load balancing algorithm, we implemented a point-centered domain decomposition method developed by Koradi [19]. At each step, this algorithm calculates a bias b_i for each domain i. When the bias increases, the domain radius and volume increase. Likewise when the bias decreases, radius and volume become smaller. We assign each atom (with position vector x) to the domain that satisfies:

$$|x - c_i|^2 - b_i = minimal, \tag{11}$$

where c_i is the center of domain i, and we calculate the new centers as the center of gravity for the atoms in each cell.

Although the Koradi algorithm is diffusive, we can run its steps independently of the application execution until it converges. Our implementation therefore treats this algorithm as a global method, and only applies the final center positions to the application. We further optimize the algorithm by parallelizing it and executing it on a sample of the atoms rather than the complete set.

In our experiments, we use a range of decompositions that exhibit different load balance properties by varying the placement of Voronoi cell centers. We evaluate all three models in Section 7.1, as well as the different load distributions and the degree of difficulty in balancing them in Section 7.2.

We used two problem sets for ddcMD, a nanowire simulation and a Condensation simulation. The *nanowire* simulation is a finite system of 133,280 Fe atoms, where the imbalance is caused by the uneven partitioning of the densely populated cylindrical body surrounded by vacuum. Atom interaction is modeled with EAM potentials. We ran the Nanowire problem on 64 processes. The *condensation* simulation is a Lennard-Jones condensation problem with 2.5e+6 particles and the interactions modeled with Lennard-Jones potentials [17], where the imbalance is caused by condensation droplets forming in some of the simulated domains. We ran the condensation simulation on 512 processes.

6.2 ParaDiS

ParaDiS [8] is a large-scale dislocation dynamics simulation used to study the fundamental mechanisms of plasticity. It is written primarily in C and uses MPI for communication between processors.

ParaDiS simulations grow in size as more time steps are executed. As such, the domain of the application is spatially heterogeneous, and the domain decomposition is recalculated periodically in order to rebalance the workload.

ParaDiS uses a 3-dimensional recursive sectioning decomposition that first segments the domain in the X direction, then in the Y direction within X slabs, and finally in the Z direction within XY slabs. Figure 2(b) illustrates the recursive sectioning decomposition.

ParaDiS uses an *empirical model* as an input to its load balancing algorithm. The application developers estimate load using timing calipers around the computation they deem most important for load balance. Data from this empirical model is then given to the recursive sectioning balancer as input. The balancer then adjusts work per process by shifting the boundaries of the sections. The magnitude of a shift is constrained by the size of neighboring domains, and the balancer will not move a boundary past the end of its neighbors. This constraint makes the ParaDiS balancer a diffusion algorithm. For our experiments, we vary the distributions of the domain by varying the *xyz decomposition* of the domain such that $x * y * z = nProcs$.

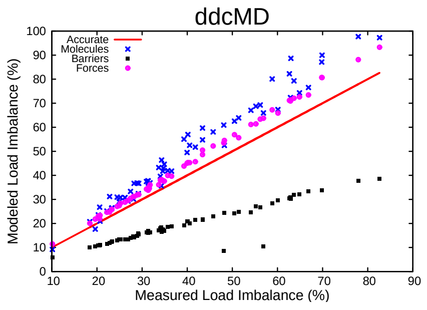
We used a highly dynamic crystal simulation input set for ParaDiS, with 1M degrees of freedom at the beginning of the simulation growing to 1.1M degrees of freedom by the end of the run. We ran this simulation on 128 processes.

7. Evaluation

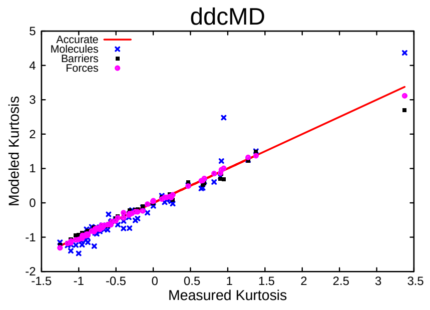
We conducted our experiments on two large scale clusters and a BlueGene/P system. For all ParaDiS experiments, we used a Linux cluster that has 800 compute nodes, each with four quad-core 2.3 GHz AMD Opteron processors, connected by Infiniband. We used a similar cluster that has 1,072 compute nodes, each with four dual-core 2.4 GHz AMD Opteron processors connected by an Infiniband interconnect for all ddcMD runs in Section 7.1. On both Linux systems, we use gcc 4.1.2 and MVAPICH v0.99 for the MPI implementation. We used a BlueGene/P system with 1,024 compute nodes with 4 32-bit PPC450d (850MHz) cores each and 64 32-bit PPC450d I/O nodes for all ddcMD experiments in Section 7.2. On this system, we use gcc 4.1.2 for our measurement framework and compile ddcMD with xlC 9.

To validate load models, we measure the actual work per process, using Libra [13], a scalable load balance measurement framework for SPMD codes. Libra measures the time spent in specific regions of an application per time step using the *effort model*. In this model, time steps, or *progress steps*, model each step of the synchronous parallel computation, and fine-grained *effort regions* within these steps model different phases of computation.

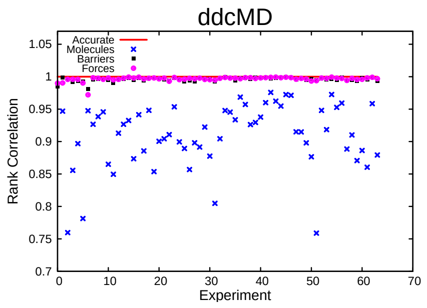
We have extended Libra's effort model so that it can be used as input to our load model. We have added an interface that allows us to query effort (load) information during execution. Using this functionality, we are able to measure the computational load on each process by summing the time spent in all effort regions. We



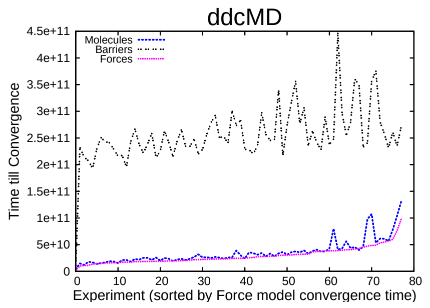
(a) % imbalance in runs.



(b) Kurtosis of load distributions.

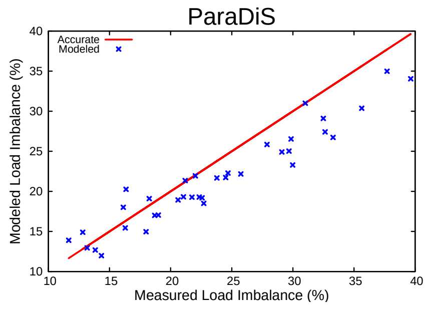


(c) Rank correlation of load distributions.

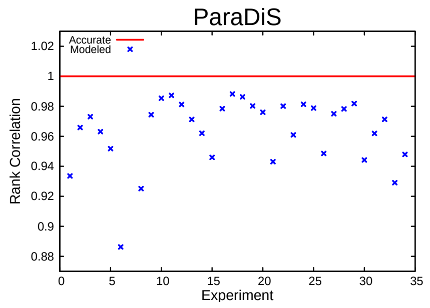


(d) Steps to converge for a diffusion algorithm using different models as input.

Figure 3. Evaluating three ddcMD models on a range of problems in ddcMD.



(a) % imbalance in runs.



(b) Rank correlation of load distributions.

Figure 4. Evaluating ParaDiS model on a range of problems in ParaDiS.

use this as an input to construct an abstract load graph, and we use this graph to compare against abstract load graphs constructed from application load models. This allows us to validate application load models against complete empirical measurements.

We integrate Libra with the rest of our load model infrastructure using P^{N}MPI [27]. This MPI tool infrastructure supports stacking of multiple independent tools that use the MPI profiling interface [6], in our case of Libra and our load model component. P^{N}MPI also supports direct communication among tools, which we exploit to exchange both topology and performance information. This mechanism enables a separation of the two components in our system, resulting in a lighter weight, more modular, extensible system. Our tool stack imposes only 3% overhead on average, making any perturbation of application behavior insignificant.

7.1 Evaluating Application Models

In this section, we evaluate the quality of *ad hoc*, developer-provided application load models by comparing them with models measured empirically using Libra. To quantitatively measure the quality of the models, we measure the accuracy with which application models capture the true load imbalance on particular timesteps. We also conduct more in-depth validation by examining how well these metrics capture the statistical moments of the true load distribution, as defined in Section 3.

For additional analysis, we validate application models with a *rank correlation* metric. Rank correlation measures how accurately the model ranks each process's load relative to the of other processes. We calculate rank correlation r of process loads between the application model M and the actual load L as:

$$r = \frac{\sum_{i=0}^{n}(m_i - \bar{m})(l_i - \bar{l})}{\sqrt{\sum_{i=0}^{n}(m_i - \bar{m})^2 \sum_{i=0}^{n}(l_i - \bar{l})^2}} \tag{12}$$

where m_i and l_i are ranks for the raw scores for the load on process i and $\bar{m}$ and $\bar{l}$ are the scores for the mean load on all processes in models M and L. To accommodate tied ranks correctly, we use Pearson's correlation coefficient [22]. We now apply our model evaluation methodology to ddcMD and ParaDiS.

Table 3. RMSE for plots in Figure 3.

	Molecules	Barriers	Forces
imbalance	15.917	25.769	16.095
kurtosis	0.444	0.079	0.057
rank corr.	0.138	0.008	0.007

Figure 3(a) demonstrates how well the three ddcMD models capture the imbalance present in the problem. Table 3 shows root mean squared error (RMSE) of the statistical moments calculated over all of our experiments. Models based on the number of molecules and force computation overestimate the imbalance in the system, while the model based on execution time excluding time spent at barriers underestimates the imbalance. Underestimating the imbalance leads to slower imbalance correction with a diffusion scheme because it is less aggressive than necessary. Alternately, overestimation pushes the limits of how much the load can be redistributed at each time step, and therefore converges as fast as a scheme that accurately estimates load, so long as the overestimation correctly captures the *relative* loads.

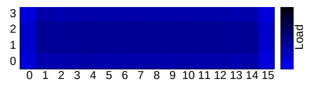
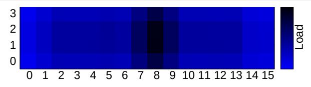
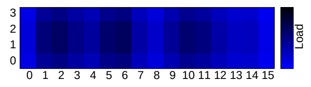
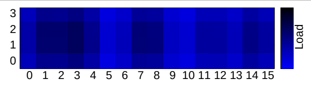
	Load on each Process	Imbal.	Steps	Diffus.	Global
(a)		1170.6	**4**	1183.4	543.6
(b)		1732.4	**77**	1061.5	828.8
(c)		1406.9	**30**	1039.4	636.7
(d)		1376.0	**35**	979.1	697.3

Table 4. Costs of balancing ddcMD nanowire simulation (in seconds).

Figure 3(b) shows how the three ddcMD models capture the kurtosis of the load distributions for each run, with corresponding RMSE shown in Table 3. The model based on the number of molecules does most poorly, because a large part of the imbalance comes from imbalanced neighbor communication, which the model omits.

Figure 3(c) shows the rank correlation between the modeled and measured distributions for each of our test cases, with corresponding RMSE shown in Table 3. Again, models based on time and force computation detect the outliers fairly well, while the model based solely on the number of particles does worse.

Overall, based on the above analysis, we expect the model based on the force computation to be the most accurate and therefore to be the most suitable for use as input to the diffusion mechanism. To validate our conclusions, Figure 3(d) shows the *steps to converge* when the diffusion algorithm uses the three models as input. We use a threshold of 12% imbalance because, as mentioned, ddcMD's best achievable balance is limited by constraints on the shape of Voronoi cells in its domain decomposition. As predicted, the model based on the force calculation is the most accurate and thus corrects the load most quickly. The model based on the number of molecules outperforms the Barrier model, in part because the former overestimates the imbalance making the diffusion scheme take more drastic measures and arrive at a balanced state sooner.

Figure 4(a) the accuracy with which the ParaDiS application model represents its load imbalance. Figure 4(b) shows the rank correlation of actual and modeled load distributions. The figures show that the model is somewhat inaccurate, which we suspected because its empirical model does measure certain major phases of the computation that are captured by Libra. The developers only measure the main force computation. Our load model in conjunction with Libra's performance attribution show that this fails to capture the behavior of communication, collision detection, and remesh phases. When we compare ParaDiS's calipers to Libra's measurements of only the force computation, the model is quite accurate. Depending on the problem, these omitted regions comprise up to 15% of the execution time. We have communicated our findings to the application developers, and intend to work with them to optimize how the application reports load to the load balancer as well as to our load model.

7.2 Cost Model Case Study

In this Section, we evaluate our model's effectiveness for modeling the efficiency of load balancing algorithms for various imbalance scenarios. We use the cost model defined in Section 5 to explain the observed performance of the global and diffusive load balancing schemes in ddcMD.

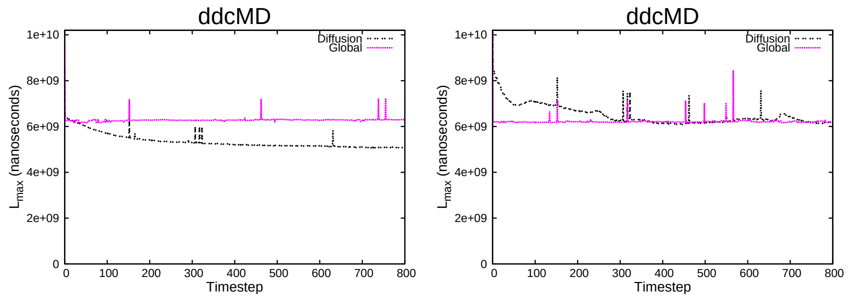
The first set of results show data from the ddcMD nanowire simulation running on 64 processors organized as a 4x16 process grid. Table 4 shows the relative load in the processor grid in the beginning of the simulation, with darker sections representing higher load and lighter blue representing lower load for the particular process. For each of these, we show the execution time without load balancing, the number of *steps* that Algorithm 5.1 predicts the diffusive algorithm will take (our metric for the difficulty of correcting the imbalance by diffusion), the execution time with the diffusion algorithm, and the execution time using the global algorithm. We run the nanowire simulation for 200 time steps. The diffusion algorithm has a cost (as defined in Equation 5) of 0.01 seconds per simulation step. The global load balancing method incurs a one-time cost of 3.5 seconds.

Example (a) in Table 4 demonstrates that diffusion can increase the execution time of the application with an almost-balanced case because it fails to find and improving the imbalance, but still incurs the cost of running the balancer. In addition to correcting the load, the global algorithm can achieve a domain decomposition that minimizes the necessary communication, something the diffusive algorithm does not do. In this case, failing to rebalance means a 2.15 times longer execution. Similarly, choosing the wrong load balancing algorithm increases the run time by 2.17 times. The remaining examples in Table 4 demonstrates that although diffusion can (eventually) decrease the computation cost of the application runs, the global algorithm is always the better choice for this ddcMD input because the diffusion algorithm takes too many steps to converge as demonstrated in Equation 8.

We further demonstrate the costs of global and diffusive schemes on 800 steps of the ddcMD condensation simulation in Figures 5 and 6. The cost of balancing algorithms themselves, as defined in Equation 5, are higher for this simulation because of the large scope of the problem. They are 0.25-0.47 seconds per step for diffusion and 95 seconds overall for the global scheme. Our cost model in Section 5 captures the additional costs of the balancing algorithms in Equation 5: the runtime of the application depends on the rate at which these load balancing schemes can correct the imbalance (or how long the application continues to run in imbalanced state) and the rate at which the application becomes imbalanced again if balanced only at the beginning of the simulation. Figure 5 demonstrates the maximum loads of processes at a given timestep in the simulation when using a global load balancing scheme and diffusion. While the maximum load shown in Figure 5 limits the length of each timestep in the simulation, the total runtime of the simulation is the integral of the curve. We list these totals Figure 6.

The initial decomposition of the problem in Figure 5(a) is relatively quick to diffuse as predicted by our *steps* metric from Algorithm 5.1 that Figure 6, row (a) shows. Figure 5(a) demonstrates that the maximum process load decreases rapidly with the diffusion algorithm, i.e., $\alpha(i)$ in Equation 8 is high. At the same time, the load does not become further imbalanced as the simulation time progresses, which means that $\beta(i)$ in Equation 10 is close to 1. Thus, we expect the overall cost of diffusion to be lower in this case than that of a global method, as Figure 6, row (a) shows.

The decomposition that Figure 5(b) shows is difficult to diffuse as predicted by the *steps* metric from Algorithm 5.1 that Figure 6, row (b) shows. Figure 5(b) demonstrates that diffusion requires over 300 steps to reduce the load of the maximum process to the average load, i.e., α in Equation 8 is high. As in case (a), the load does not become further imbalanced as the simulation time progresses; so β in Equation 10 is again low. We expect the overall cost of diffusion to be higher in this case than that of a global method, as Figure 6, row (b) shows.



(a) Easy to diffuse. (b) Difficult to diffuse.

Figure 5. L_{max} for diffusion and a global scheme in different imbalance scenarios.

	Imbalanced	Diffusion			Global	
		steps (5.1)	cost /step	exec. time	total cost	exec. time
(a)	5104.26	71	.025	4090.54	95	5115.87
(b)	6444.54	344	.047	5231.01	95	5063.30

Figure 6. Costs of balancing for ddcMD condensation simulation (in seconds).

Generally, a global method is preferred when the imbalance would take many steps to correct using diffusion, and when the cost of the global method is less than that of continuing to run the simulation in an imbalanced state. More precise guidelines than these are both application dependent and problem dependent. Our future work will include deriving the coefficients for our models by sufficiently sampling the space and using available modeling techniques [20]. However, predictive models are beyond the scope of this paper. Overall, understanding the costs of the load balancing algorithms is important in selecting the appropriate one for each imbalance scenario.

8. Related Work

Previous work has focused on load measurement and finding sources of imbalance. Efficient, scalable measurement of load [16, 32] identifies whether load imbalance is a problem for a particular application. Imbalance attribution [31] provides insight into the source code locations that cause imbalance. Our load model takes advantage of existing tools [13] and their measurements, and combines them with knowledge of the simulated application domain topology. This allows better understanding of computational load in terms of an application's particular work units.

Many applications that suffer from load imbalance implement their own load balancing algorithms. The algorithms are usually tightly coupled with application data structures and cannot be used outside of the application. Some rely heavily on geometric decomposition of the domain (i.e., hierarchical recursive bisection [8]). AMR applications can order the boxes according to their spatial location by placing a Morton space filling curve [21] through the box centroids to increase the likelihood that neighboring patches reside on the same process after load balancing [36]. N-body simulations either explicitly assign bodies to processes or indirectly assign bodies by assigning subspaces to processes using orthogonal recursive bisection [4], oct-trees [28, 35], and fractiling [2]. In all of these cases, the application developers must construct an *ad hoc* model of per-task load. These estimates are frequently inaccurate because they omit a significant subset of computational costs, or they fail to consider the platform. Our load model enables evaluation of the applicaton models, therefore ensuring that the computation costs are assigned appropriately prior to being used to correct the imbalance.

Another common approach to load balancing uses suites of partitioners that work with mesh or graph representations of computation in the applications (e.g., ParMetis [26], Jostle [33], Zoltan [12], DRAMA [3], PLUM [23]). Users of these partitioners must supply information about the current state of the application and the system, which again must be application specific. As a result, they may provide inaccurate or incomplete information. Further, partitioners do not have sufficient information to decide when to load balance, placing further burden on the application. Our load model enables actionable evaluation of load imbalance, and can potentially help an application to decide when to rebalance and whether the repartitioner is a good balancing algorithm for a particular type of imbalance.

Charm++ [1, 5, 7, 18, 37] implements a measurement-based load balancing framework that records the work represented by objects and object-to-object communication patterns. Based on the RTS measurements, the load balancer may migrate the objects between process queues. While this approach may work well when the application developers can decompose their computation into independent objects, many applications cannot. In other cases, Charm++ can overdecompose the domain [24] and then balance load by moving virtual processors from overloaded physical processors to the underloaded ones. This approach can impose extra communication overhead for tightly coupled applications. In more recent work, Charm++ explores hierarchical approaches to load balancing [14]. Our model's ability to evaluate the load in an actionable manner could therefore provide a sound basis to choose the best level at which to balance the load.

9. Conclusions

We have presented a novel load model that captures an application's topology, load, and execution time. Our load model establishes a mapping between application elements and computation costs while maintaining information on dependencies between model elements. Our load model enables an application-independent representation of load distribution and can form the basis for a new generation of generic, yet application- and topology-aware load balance tools. We have shown that our topology-aware approach overcomes deficiencies of conventional statistical load metrics, which fail to represent topology information. Using our topology-aware load model we have provided a new set of metrics characterize load distribution more accurately.

We have demonstrated the effectiveness and versatility of our load model on several case studies. We have provided a mechanism to evaluate and contrast several application-provided models. We have used our topology-aware load model to analyze the load imbalance in the application at a given point in time. Finally, we evaluated the ability of available load balance schemes to correct imbalances. In all experiments, adding the topology information to the load data was critical to understanding and analyzing the application's load behavior.

References

[1] T. Agarwal, A. Sharma, and L. V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Intl. Parallel and Distributed Processing Symposium*, April 2006.

[2] I. Banicescu and S. Flynn Hummel. Balancing processor loads and exploiting data locality in N-body simulations. In *ACM/IEEE Conf. on Supercomputing*, 1995.

[3] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J. M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw. Dynamic load-balancing of finite element applications with the DRAMA library. *Applied Mathematical Modelling*, 25(2), 2000.

[4] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(5), 1987.

[5] A. Bhatelé, L. V. Kalé, and S. Kumar. Dynamic topology aware load balancing algorithms for MD applications. In *ACM SIGARCH Intl. Conf. on Supercomputing*, 2009.

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Intl. Journal of High Performance Computing Applications*, 14(3), 2000.

[7] R. K. Brunner and L. V. Kalé. Handling application-induced load imbalance using parallel objects. *Parallel and Distributed Computing for Symbolic and Irregular Applications*, 2000.

[8] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable line dynamics in ParaDiS. In *ACM/IEEE Conf. on Supercomputing*, 2004.

[9] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1), 1999.

[10] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 1989.

[11] B. R. de Supinski, M. Schulz, V. V. Bulatov, W. Cabot, B. Chan, A. W. Cook, E. W. Draeger, J. N. Glosli, J. A. Greenough, K. Henderson, A. Kubota, S. Louis, B. J. Miller, M. V. Patel, T. E. Spelce, F. H. Streitz, P. L. Williams, R. K. Yates, A. Yoo, G. Almasi, G. Bhanot, A. Gara, J. A. Gunnels, M. Gupta, J. Moreira, J. Sexton, B. Walkup, C. Archer, F. Gygi, T. C. Germann, K. Kadau, P. S. Lomdahl, C. Rendleman, M. L. Welcome, W. McLendon, B. Hendrickson, F. Franchetti, S. Kral, J. Lorenz, C. W. Uberhuber, E. Chow, and U. Catalyurek. BlueGene/L applications: Parallelism On a Massive Scale. *Intl. Journal of High Performance Computing Applications*, 22(1), 2008.

[12] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, J. Teresco, J. Faik, J. Flaherty, and L. Gervasio. New challanges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3), 2005.

[13] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Scalable load-balance measurement for SPMD codes. In *ACM/IEEE Conf. on Supercomputing*, 2008.

[14] E. M. Gengbin Zheng, Abhinav Bhatele and L. V. Kale. Periodic Hierarchical Load Balancing for Large Supercomputers. *Intl. Journal of High Performance Computing Applications*, 2010.

[15] J. N. Glosli, K. J. Caspersen, J. A. Gunnels, D. F. Richards, R. E. Rudd, and F. H. Streitz. Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability. In *IEEE/ACM Conf. on Supercomputing*, Nov. 2007.

[16] K. A. Huck and J. Labarta. Detailed load balance analysis of large scale parallel applications. In *Intl. Conf. on Parallel Processing*, 2010.

[17] J. E. Jones. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Royal Society of London Proceedings Series A*, 106:463–477, Oct. 1924.

[18] G. A. Koenig and L. V. Kalé. Optimizing distributed application performance using dynamic grid topology-aware load balancing. In *IEEE Intl. Parallel and Distributed Processing Symposium*, 2007.

[19] R. Koradi, M. Billeter, and P. Gntert. Point-centered domain decomposition for parallel molecular dynamics simulation. *Computer Physics Communications*, 2000.

[20] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Intl. Symposium on Principles and Practices of Parallel Programming*, 2007.

[21] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. *IBM technical report*, 1966.

[22] J. L. Myers and A. D. Well. Research design and statistical analysis (2nd ed.). *Lawrence Erlbaum Associates Publishers*, page 508, 2003.

[23] L. Oliker and R. Biswas. PLUM: parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distr. Computing*, 1998.

[24] E. R. Rodrigues, P. O. A. Navaux, J. Panetta, A. Fazenda, C. L. Mendes, and L. V. Kale. A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model. In *Intl. Symposium on Computer Architecture and High Performance Computing*, 2010.

[25] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Intl. Euro-Par Conf. on Parallel Processing*, 2000.

[26] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *ACM/IEEE Conf. on Supercomputing*, 2000.

[27] M. Schulz and B. R. de Supinski. P^{N}MPI tools: a whole lot greater than the sum of their parts. In *ACM/IEEE Conf. on Supercomputing*, 2007.

[28] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *ACM/IEEE Conf. on Supercomputing*, 1993.

[29] F. Streitz, J. Glosli, M. Patel, B. Chan, R. Yates, B. de Supinski, J. Sexton, and J. Gunnels. Simulating solidification in metals at high pressure: the drive to petascale computing. *Journal of Physics: Conf. Series*, 46, 2006.

[30] F. Streitz, J. Glosli, M. Patel, B. Chan, R. Yates, B. de Supinski, J. Sexton, and J. Gunnels. 100+ TFlop Solidification Simulations on BlueGene/L. In *IEEE/ACM Conf. on Supercomputing*, Nov. 2005.

[31] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *ACM/IEEE Conf. on Supercomputing*, 2010.

[32] N. R. Tallent, J. M. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable fine-grained call path tracing. In *ACM/SIGARCH Intl. Conf. on Supercomputing*, 2011.

[33] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12), 2000.

[34] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2), 1997.

[35] M. S. Warren and J. K. Salmon. A parallel hashed Oct-Tree N-body algorithm. In *ACM/IEEE Conf. on Supercomputing*, 1993.

[36] A. M. Wissink, D. Hysom, and R. D. Hornung. Enhancing scalability of parallel structured AMR calculations. In *ACM SIGARCH Intl. Conf. on Supercomputing*, 2003.

[37] G. Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Dept. of Comp. Science, U. of Illinois at Urbana-Champaign, 2005.